Compact routing on low earth orbit satellites

Stefan Rotarus

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

Spring 2019

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisors
Cristina Basescu
Kelong Cong
EPFL / DEDIS

Contents

1 Introduction **1**

2 Goals and Motivation **1**
2.1 A new satellite network 1
2.2 Faster routing at small costs 1

3 Satellite simulation challenge **2**
3.1 Chosing a simulator 2
3.2 Orbital Parameters 3
3.3 Simulation solution 4

4 Network implementation **5**
4.1 Physical constraints 5
4.2 Implementation 8
4.3 Compact Routing 8

5 Results **10**
5.1 Space 10
5.2 Preprocessing time 12
5.3 Routing Distance 13

6 Limitations and Future Work **14**
6.1 High-Frequency Trading 14
6.2 Satellite Links 14
6.3 Next stages of Starlink 14
6.4 Ground Stations 14

7 User Guide **15**
7.1 leo_satellites.py 15
7.2 network_implementation.py 16

1 Introduction

Recently, a couple of companies decided to invest into low earth orbit satellite (LEOs) to achieve very low latency on internet transmissions. The first purpose of such technologies will be to achieve high-frequency transmissions, in particular for financial operations.

The second and more altruistic goal would be to provide better latency to remote areas, cargo boats and many other places where a terrestrial infrastructure appears hard to build.

The architecture of such satellites networks appears to be very regular and offers a low amount of unpredictable events. For this reason, we decided to simulate the efficiency of Compact Routing algorithms in order to achieve smaller routing tables and fewer computations than a Dijkstra algorithm. Of course, this has a cost, and we are here looking at how the total routing distance will be affected by the use of Compact Routing.

2 Goals and Motivation

2.1 A new satellite network

SpaceX began in 2015 its researches for the implementation of it own LEOs Network, called *Starlink*. The deployment of this constellation started this year and is meant to continue until 1600 satellites are deployed over 32 orbital plans [2]. Therefore, another 2825 will be added on different orbital plans, meant to cover the most inhabited regions on earth in an optimal way. Considering this is one of the most complete projects ever realised on LEOs and it is not yet fully deployed, we decided to make our simulations on this concrete network.

2.2 Faster routing at small costs

A few recent papers tried to estimate the latency offered by this new constellation and arrived at great results [6]. We decided to simulate a Compact Routing algorithm using similar parameters, compute the size of routing tables, the execution time and the routing distances provided by these schemes [7]. The main goal here is to show we can achieve smaller routing table and execution time with small loss of performance, in terms of routing.

3 Satellite simulation challenge

3.1 Chosing a simulator

The first big step was to find the required tools to simulate a LEO network. The following paragraphs are a list of the simulators I found with some explanations on each. These in formations are specially important to avoid wasting time on the wrong softwares in future projects.

SaVi

SaVi[13] allows to easily generate a LEO constellation, based on the parameters provided by SpaceX and to export it as a TCL script containing the precise orbital paramters for each satellite. It also shows the movement of the constellation and its coverage projected on earth. Unfortunately, there is no easy way to export or retrieve the position (longitude, latitude) of the satellites.

NS2 and NS3

NS-2[8] provides support for routing simulation. Unfortunately NS-2 is an old software with a very poor documentation. The part regarding satelliteâĂŹs simulation is no more available in the documentation. Moreover, the installation appears really hard. After trying most of the installation method, there was no way of solving the multiple errors that occurred, mainly due to old libraries.
NS-3[9] is the newer version offering a very large documentation. It seems really complete and even offers a graphic display of the graphs using Qt. Unfortunately, the installation of the program was a huge waste of time in this project. The multiple installation attempts took a couple of days, and failed on my computer even with the great help of my supervisors. After another couple of days we managed to install it on a lab's computer running Ubuntu.
Once installed, the software appears very hard to use with the knowledge I have in networks. It appears to be designed for network professionals looking for very complete simulations, therefore somehow an overkill in our case.

Pyephem

Pyephem[11], used in a previous paper on LEOs[3], was by far the best solution I found. It is a python library that allows simulation on multiple astronomical bodies. Based on the orbital parameters, Pyephem returns the longitude, latitude and altitude of a satellite. This solution requires deeper understanding concerning the Keplerian elements, that will be detailed in the soon for the comfort of the reader.

Skyfield

Skyfield[12] is a new version of Pyephem that doesn't include the implementation errors of its predecessor. Unfortunately, only the simulation of already existing satellites is made easy for the user. A solution I considered would be to conceive a false Two-Line-Elements file, a format that contains all the orbital parameters we need. This is a very strict file format, requiring a deep understanding of orbital parameters. It appears to be strongly advised in the astronomic community to avoid creating unofficial TLE files.

3.2 Orbital Parameters

In order to understand the simulations, the reader needs to be familiar with what the main orbital parameters mean. They are used to fully describe the movement of a satellite around the earth, and some of them could be expressed differently in academic papers or soft-wares. A tutorial published on AMSAT's webpage [10] provides a good intuitive understanding for them. Here is a quick explanation of the most important ones for future understanding of this project.

Orbital Inclination is the angle between the equator and the disk formed by the satellite's on a an orbit. For example, an orbital inclination of 0 will mean the satellite is always orbiting above the equator. We mainly considered in our simulations the first phase of launching for the Starlink constellation, with an Orbital Inclination of 53°[2]

Phase Offset - between orbital planes - is a number between 0 and 1 that indicates when the satellite n on plane p passes the equator, compared with satellite n on plane p+1. If the phase offset is 0, both pass the equator at the same time. If it's 1, n on plane p passes the equator at the same time as n+1 on plane p+1. The goal of this parameter is to maximize the minimal distance between any two satellites, to avoid collisions. We use here the value calculated in Mark Handley's paper [6], namely 5/32.

Coverage Angle is the angle in which a satellite is considered as reachable from the ground. In our case, we can deduce for SpaceX's technical attachments [2] that we face a 40°angle.

Here is a quick summary of **all** the data that could have been obtained concerning the constellation or deduced by simulations in previous papers I had access to.

Planes Num.	Sats/Plane	Altitude	Inctination	Offset	Coverage
32	50	1150km	53°	5/32	40°

3.3 Simulation solution

SaVi configurations

We choose to first use SaVi [13] in order to obtain a visual representation. Be careful to include GeomView [5] in order to have a clean 3D representation of our satellite link. Once there, you can import the satellite files provided with this report (.tcl files). If you need to simulate other configuration of satellites for LEOs, select Constellation, choose Ballard Rosette and enter the parameters mentioned before.
In 1 you can see a representation of the Starlink constellation obtained with SaVi.

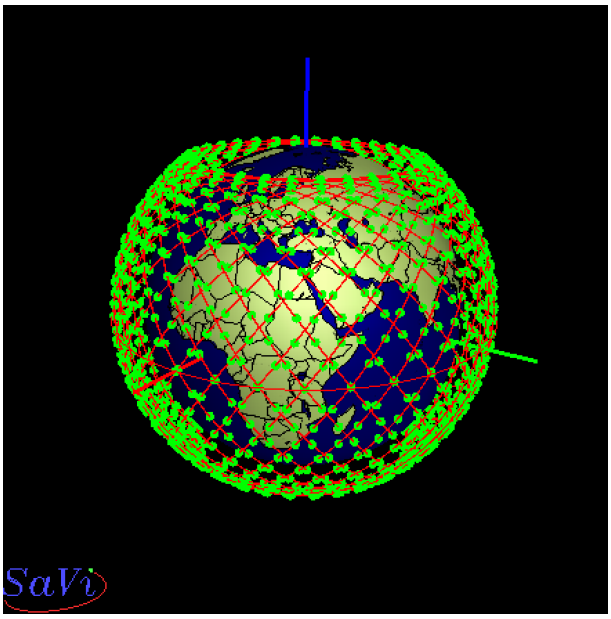

Figure 1: Stage 1 of Starlink Constellation

Once the orbital parameters have been choose, one can export the satellites parameters into a .tcl file. This file will contain for each satellite, its own six orbital parameters, as well as some graphical parameters we can ignore in the next steps.

Pyephem conversion

The python implementation simply reads the .tcl file and sets pyephem orbital values accordingly. Unfortunately, both simulator do not use the exact same set of orbital values, requiring some mathematical conversions. The following table is meant to give a translation between these orbital parameters. They are given in the order they can be found in the tcl files coming from SaVi.

SaVi	**Keplerian Elements**[10]	**Pyephem Representation**
Semi-Major Axis (meters)	Mean Motion (revo./day)	n (revo./day)
Eccentricity (°)	Eccentricity (°)	e (°)
Inclination (°)	Orbital Inclination (°)	inc (°)
Longitude of Ascending Node (°)	Right Asending. Node (°)	raan (°)
Arg. Periapsis (°)	Arg. Perigee (°)	ap (°)
Time to periapsis (seconds)	Mean Anomaly (degrees)	m (degrees)

Parameters conversion

The implementation contains a file named "geometryfunctions.py" that includes all the geometrical transformations from SaVi to Pyephem parameters. At this point, we have everything we need to run Pyephem, and retrieve the position of each satellite.

4 Network implementation

4.1 Physical constraints

Laser links

The satellites are connected to one another though laser links. According to the European Space Agency, the range of such links is around 50'000km [1]. The entire satellite constellation is at the surface of a $D_{constellation}$ diameter sphere. $D_{constellation} = 2 * R_{earth} + Altitude = 15'042km$
The only physical limitation will be the earth curve itself. The following result was obtained with the graphical calculator GeoGebra. One can easily see this limitation and deduce we can not have laser transmission with satellites that are at more than 7'993km at figure 2.

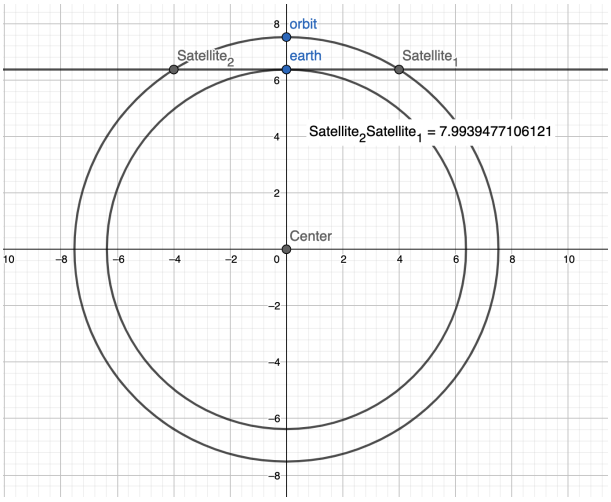

Figure 2: Earth curve limitation

Another preliminary question was whether two satellites could be reachable by an earth station, but not reachable via laser link. As a quick reminder, a satellite is considered as reachable from a ground station, it is in a 40°angle from the vertical line formed by the ground station and the center of earth.
Once again, we deduced using GeoGebra that two satellites reachable from a ground station are at most at 1'836km from one another in 3.

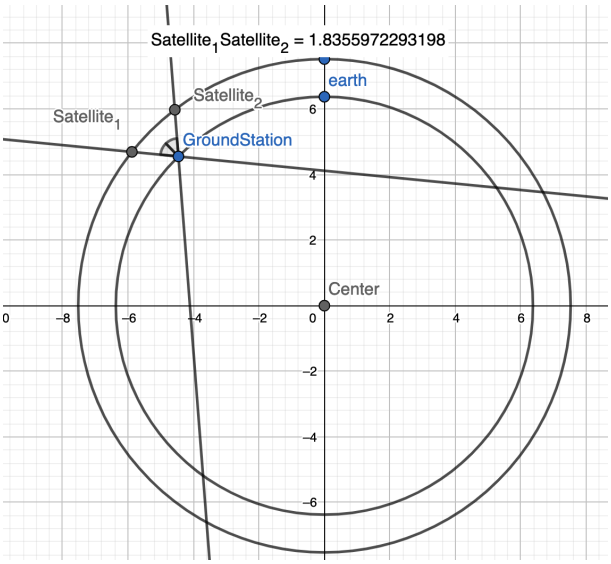

Figure 3: Ground station reachability

Number of links

The satellites from the Starlink contellation will all have five optical links in order to communicate with their neighbours [2]. ESA informs us that a satellite can bring up its optical link to communicate with other satellites in under a minute, but not in negligible time, in the similar European Data Relays System technology [4]. Therefore, it is important to choose correctly five satellite links to minimize the connection distances between satellites.
Starting from a satellite, only the ones on the same orbital plane and on the closest orbital planes will be constantly in range. Considering satellite n on orbit p, it appears obvious to choose satellites n+1 and n-1 on p, to provide good connectivity on this axis. Now, the goal is to maximize connectivity on the perpendicular axis. The optimal solution here are satellite n on p+1 and satellite n on p-1. Any other satellite chosen on p or p+1 will lead at a link that is almost parallel to the ones we already have.
The hard question now is how to chose the last link. Fortunately, Mark Handley's work [6] provides a good answer to this situation, namely choosing the next closest satellite, and updating it frequently. As surprising as it could appear, this solution increases well the connectivity in the graph, even if the link is often updated under a one minute delay. With current technology, the delay for updating links between very close satellites is reduced.
The reader can convince himself of the validity of this geometrical choice using the figure 4

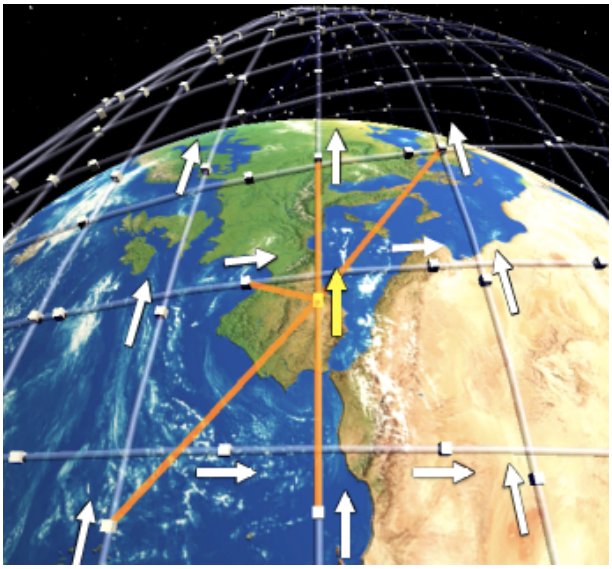

Figure 4: Earth curve limitation

4.2 Implementation

Once the position of all the satellites has been retrieved, as explained in 3.3, one can easily calculate the distance between all the satellites. The first step we choose was to apply the haversine formula, in order to obtain the distance between the two points at the surface of the sphere formed by our constellation.
The calculations are based on the latitude and longitude of the two satellites. Here $R = R_{earth} + Altitude$.

$$d = 2R\sin^{-1}(\sqrt{\sin^2\frac{Lat_2 - Lat_1}{2}) + \cos(Long_1)\cos(Long_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2})})$$

Once this distance has been computed, we simply apply the formula to calculate the chord of a sphere, namely the direct distance between the two points passing into the sphere, and not on the surface.

$$chord = 2R * \sin(\frac{d}{2R})$$

NetworkX
At this point we obtain a matrix containing the distance between any two satellites in the constellation. Therefore, we chose to implement this structure as an undirected graph, using the Python Library *NetworkX*. Here the satellites will be represented by nodes and the distance between them by edges of corresponding weight. This implementation attributes a node number between 0 and $Size_{Constellation} - 1$ to each satellite.

As mentioned in subsection 4.1, only five laser links are possible for each satellite. Therefore, the graph using to represent the constellation will have a degree of five for each node, where the corresponding edges are chosen according to the real satellite links described in subsection 4.1.

Tests For test purposes and quick understanding of how the future algorithms work, we also chose to implement a couple of simpler graphs.

4.3 Compact Routing

Most of the papers studying Low Earth Orbit constellations considered by default Dijkstra as the routing algorithm in their simulations. We wanted to show that using Compact Routing Algorithms [7], we could obtain smaller routing tables, lower computation time at small cost in terms of routing distance.

For a quick and intuitive understanding, here is how our implementation of compact routing works.
Levels: All the nodes are randomly assigned a level between 0 and some constant k-1. Each node is at least at level 0. The probability for it to be at level i+1 knowing it is at level i is $n^{-1/k}$ where n is the total number of nodes and k the number of levels.
Bunches: From one node, we run the Dijkstra algorithm. We start a counter at level 0. We add every level-0 node discovered to the Bunch, until we encounter a node at level 1. Then we update the counter, and start adding all level-1 nodes to the Bunch, until we encounter a node at level 2. We do this until discovering a node at the maximum level, i.e k-1. Finally, we add all the nodes at maximum level to the bunch. **Clusters:** Each time a node N1 has a node N2 in its bunch, it means the node N1 is in N2's clusters. Clusters and bunches are therefore computed at the same time in our implementation.

In figure 5 you can find this algorithm as stated in the theoretical paper [7]. Here B(v) represents the Bunch of node v.

```
algorithm prepro_k(V, δ)
A_0 ← V ; A_k ← ∅
for i ← 1 to k − 1
    let A_i contain each element of A_{i−1},
    independently, with probability n^{−1/k}.
for every v ∈ V
    for i ← 0 to k − 1,
        let δ(A_i, v) ← min{ δ(w, v) | w ∈ A_i},
        and p_i(v) ∈ A_i be nearest to v, i.e. δ(p_i(v), v) = δ(A_i, v).
    δ(A_k, v) ← ∞
    let B(v) ← ∪_{i=0}^{k−1} { w ∈ A_i − A_{i+1} | δ(w, v) < δ(A_{i+1}, v)}.
```

Figure 5: Prepossessing Algorithm

Based on this, we implemented the algorithm at 6 giving distances between any two satellites based on their bunches.

```
algorithm dist_k(u, v)
w ← u ; i ← 0
while w ∉ B(v)
    i ← i + 1
    (u, v) ← (v, u)
    w ← p_i(u)
return δ(w, u) + δ(w, v)
```

Figure 6: Distance Algortihm

More details can be found on the commentaries of the code and the guide to use, but are not in my opinion relevant here. Note that many tests on different graphs have been done and explained in the code, and they could be useful as a quick understanding guide for the user.

Results Analysis

My first idea was to use the *Matplotlib* that worked well on the test graphs. Unfortunately, the reader needs to know this library doesn't work on high quantity of data, therefore I used and strongly recommend *Bokeh* diagram, that is very similar but way more complete. It was specially design for that analysis, therefore there should not be a problem in using it for even very large constellations.

5 Results

The main advantage of Dijkstra on Compact routing is the certainty to obtain the smallest possible path between node. Compact routing of level k will produce routing distances being at most 2k-1 time bigger than Dijkstra, but often better. The more regular the network is, the better will be the results. Here we tried to show that on a grid-shaped network the values are way better than the 2k-1 stretch. We will firstly state here the advantages of compact routing, namely the size of the routing tables and the preprocessing time along with the measures made on them.
The level chosen for the following simulations is always k=3, choice motivated by the good compromise it offers between small routing tables and not that big difference in terms of routing distances.

5.1 Space

(Dijkstra) algorithm requires a routing table of $O(n^2)$ space. The theoretical bound offered by our implementation of compact routing [7] is $O(kn^(1/k))$. Once again, k is the level chosen and n the number of nodes.

Our simulation was made considering the first step of the Starlink Constellation, where n=1600 nodes/satellites. At figure 7 you can find comparison of the routing table's size obtained with compact routing, against Dijkstra. Here we consider Dijkstra requires exactly n size tables for each satellite, that seems a good assumption. The figure 8 shows the average routing table size on multiple runs.

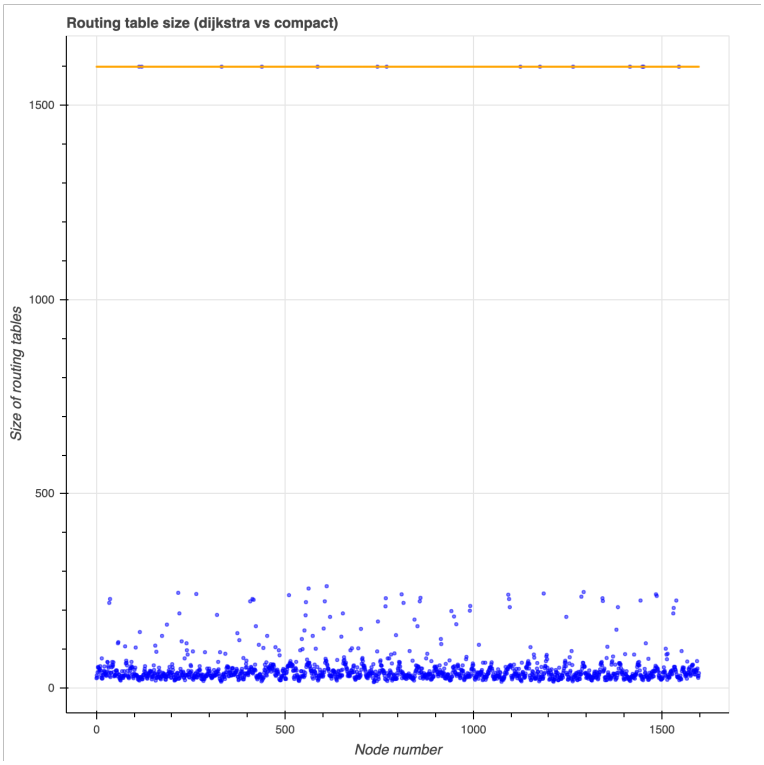

Figure 7: Routing tables size comparison
Blue: Compact Routing, Orange: Dijkstra

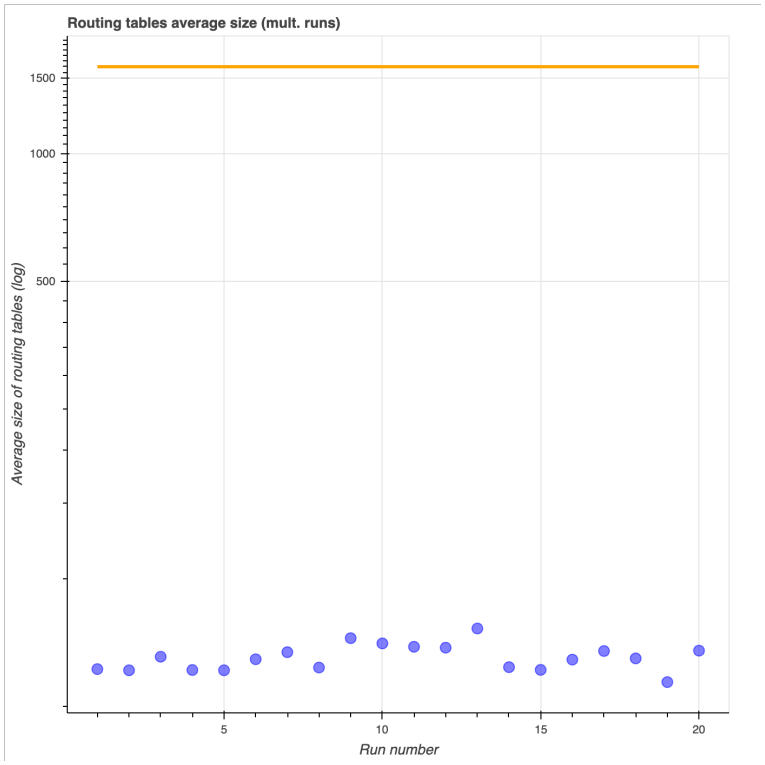

Figure 8: Average routing table size
Blue: Compact Routing, Orange: Dijkstra

Conclusion

Compact routing ensures routing table sizes around 25 times better than Dijkstra. However, the average size of the routing tables on multiple runs, even of the same graph, could increase twofold. This is due to the fact that the nodes levels are generated at random.

5.2 Preprocessing time

The optimal prepossessing time offered by this Compact Routing algorithm for **all the nodes** is $O(kmn^{(}1/k))$, where m is the number of edges, where Dijkstra will be $O(mn)$.
In our implementation, we stop Dijkstra when all the nodes at the last level have been found. Therefore, it should be always quicker than Dijkstra. Unfortunately, practical results are not ideal on this point. The Dijkstra algorithm is implemented by default in the *NetworkX* library used is strongly optimized by excellent Python programmers. Therefore, the execution time for a n=1600 graph is around 15 seconds.

On the other side, my skills as a Python programmer were somehow limited, as I had to learn the language while coding the project. Thanks to *cProfile* solve most of the implementation problems, using Dictionaries, List and Heaps at the correct place. However, the running time in our case remains around 40 seconds. The main explanation to this are the multiple calls made by the algorithm to the attributes of the nodes of the graph, calls that are not made by native Dijkstra. This design was mainly conceived as intuitive and easy to use, but appears to be less efficient.

5.3 Routing Distance

In terms of routing distance, the upper stretch will be $D_{Compact} = (2k - 1) * D_{Dijkstra}$. Fortunately, as we expected on this very regular graph, the compact distances are, for most of them, way smaller than this bound, as we can see on figure 9. Most of the results, but not all, are such as $2 * D_{Dijkstra} \leq D_{Compact} \leq 2 * D_{Dijkstra}$. It this figure, each point represents a pair of satellites. As there are nearly 2.5 millions such pairs, we chose 1'000 of them at random.

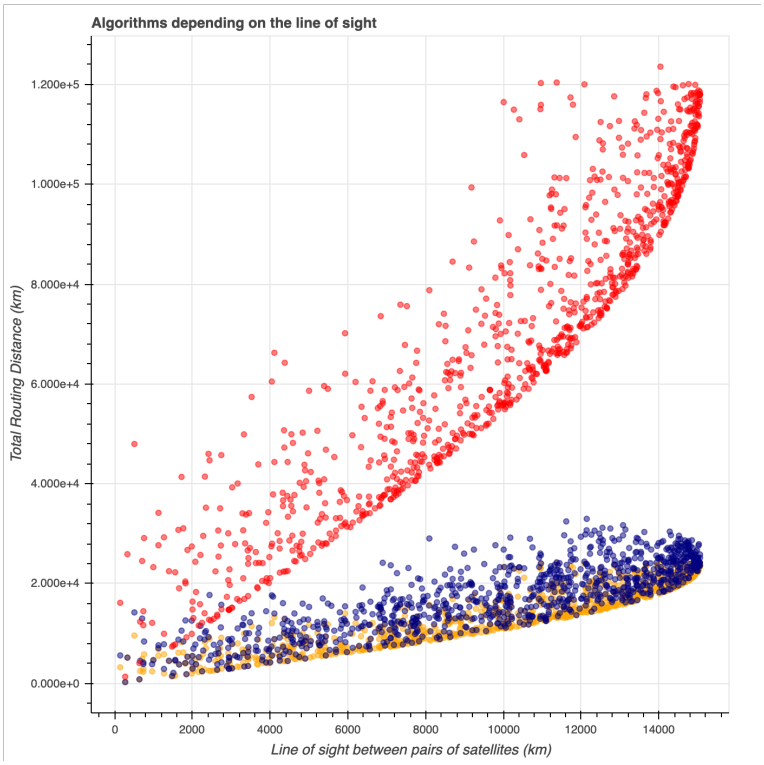

Figure 9: Routing Distances Comparison
Blue: Compact Routing, Orange: Dijkstra, Red: Bound

6 Limitations and Future Work

6.1 High-Frequency Trading

The speed of light in vacuum is only 47% higher than the speed of light in optical fiber. Considering we will have here more straight lines and fewer detours than terrestrial networks, we could also consider this improvement could increase. Previous simulations [6] state an improvement of roughly 70% in terms of latency. Even we have results way better than the upper bound offered by compact routing, taking a path that is two times dijkstra makes us lose all the benefits due to higher speed. Of course, this is only a limitation if we want to achieve the best possible latency.

6.2 Satellite Links

As a future work, it could be interesting to see if the choice of the 5th satellite link, as stated in 4.1 is the best one. It appears intuitive and has been simulated in Prof. Mark Handley's researches [6] but we can not be certain it is the best option. More simulations would let us know if there are better choices.

It will also be useful to simulate the link changes. This simulation is in theory very easy to do with the already implemented code, but relies at this point on the assumption new laser links are created instantaneously, which is wrong. Such a simulation should take into account the time to establish new laser links, depending on the distance between satellites and the precise technology used.

6.3 Next stages of Starlink

There are three main reasons we concentrated our researches only on the first stage of the Starlink constellations: the previous articles made their simulations on it, therefore it was easy to find some of the information required for this project that were not public; we have no real knowledge on what the links between the future layers of the constellation will be; finally, as part of the satellites have already been launched, we work on a very concrete case.
It would be really interesting for a future work to simulate Compact Routing on the final satellite constellation.

6.4 Ground Stations

This project includes only transmissions between the satellites in space. A more complete simulation should include the ground stations, and measure

the evolution of the laser-links to the ground. Once again we need here to have a better idea on how such links to the stations work.

7 User Guide

This project has been conceived as a simulation. The main simulation has been conducted on the first stage of the Starlink constellation, however it has been made easy for any user to make simulations on any satellite constellation, if all the constellation's satellites are at the same altitude.
As a first step, clone the project from the following repository `https://github.com/dedis/student_19_leos.git`.

NB: SaVi
If you need to simulate your own constellation, please refer to 3.3-SaVi configurations, and follow the steps described there. In case you generated a .tcl file with names for satellites, they are at the end of the file. Please delete them before using the file (only the lines defining names, nothing else, not even the commentaries).

Existing tcl files
Along with the project comes a .tcl file. *SpaceX_constellation1_1150km* contains the the constellation described in the report, namely the stage one of Starlink.

7.1 leo_satellites.py

This file has been designed for pure satellites simulation. It does all the work from the import file to the creation of the NetworkX graph but does no computation on it.

The first four constants are used to make sure the files imported from SaVi correspond to the constellation you want to simulate. Please modify them accordingly with the correct NUMBER_OF_ORBITS (i.e the number of orbital planes), SATELLITES_PER_ORBIT, ALTITUDE and TCL_FILE_NAME.
EPOCH is the launching time of your constellation. OBSERVATION_DATE is the date and time at which you would like to retrieve the position.

From there *create_spaceX_graph* will generate a graph with five edges per node and *create_full_mesh* will generate a full mesh graph. You are not required to call manually these functions to plot the graphs we generated in this report.

7.2 network_implementation.py

This file contains the compact routing algorithm, Dijkstra as well as the plotting functions. The user simply has to launch plot_routing_table_sizes(), plot_average_routing_tables() well plot_dij_vs_compact_large() with the desired number of nodes in the constellation to retrieve the simulations.

If no modification was made to the program, simply run network_implementation.py with **Python 3**.

References

[1] *Advanced Laser Technology On the Way to Space*. URL: https://artes.esa.int/news/advanced-laser-technology-way-space.

[2] Mihai Albulet. *SPACEX V-BAND NON-GEOSTATIONARY SATELLITE SYSTEM*. URL: https://licensing.fcc.gov/myibfs/download.do?attachment_key=1190019.

[3] Waqar Aqeel Debopam Bhattacherjee. "Gearing up for the 21st century space race". In: (2018).

[4] *ESA-European data relay satellite system*. URL: https://artes.esa.int/edrs/overview.

[5] Geomview. *Geomview*. URL: http://www.geomview.org/.

[6] Mark Handley. "Delay is Not an Option: Low Latency Routing in Space". In: (2018).

[7] Uri Zwick Mikkel Thorup. "Approximate Distance Oracle". In: (2001).

[8] *Network Simulator 2*. URL: https://www.isi.edu/nsnam/ns/.

[9] *NS3 - Network Simulator 3*. URL: https://www.nsnam.org/.

[10] *Orbital Parameters meaning*. URL: https://www.amsat.org/keplerian-elements-tutorial/.

[11] *Pyephem*. URL: https://rhodesmill.org/pyephem/.

[12] *Skyfield*. URL: https://rhodesmill.org/skyfield/.

[13] Lloyd Wood. *Satellite Visualisation Tool*. URL: https://savi.sourceforge.io/.